DOCUMENT RESUME

ED 318 797 TM 014 923

AUTHOR Recker, Margaret M.; Pirolli, Peter
TITLE A Model of Self-Explanation Strategies of Instructional Text and Examples in the Acquisition of Programming Skills.
PUB DATE Apr 90
NOTE 14p.; Paper presented at the Annual Meeting of the American Educational Research Association (Boston, MA, April 16-20, 1990).
PUB TYPE Reports - Research/Technical (143) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Comparative Analysis; *Computer Science Education; Elementary Secondary Education; *Learning Strategies; *Problem Solving; *Programing; *Protocol Analysis; *Skill Development
IDENTIFIERS LISP Programing Language; *Self Explanation Strategies

ABSTRACT

Students learning to program recursive LISP functions in a typical school-like lesson on recursion were observed. The typical lesson contains text and examples and involves solving a series of programming problems. The focus of this study is on students' learning strategies in new domains. In this light, a Soar computational model of self-explanation strategies of instructional text was examined. The subjects, 12 students with little or no computer programming experience, proceeded through a series of lessons in the LISP Tutor program on recursion. Subjects, who were asked to think aloud, were videotaped as they read through an instructional booklet prior to solving problems using the Tutor. Subjects were also asked to explain examples to themselves. After reading their texts, subjects worked through 12 recursion problems with the LISP Tutor. Subjects were then divided into groups of "good" and "poor" performers. The verbal protocols were segmented into elaborations, which are pause-bound utterances and are not a first reading of the text. Each elaboration was categorized in terms of the instructional content to which it referred and to one of six types of comments. Sequences of related elaborations were coded into episodes (or macro-codings). Results indicate a strong correlation between types and contents of elaborations made and subsequent problem-solving performance. Self-explanations of "good" students were much more structured into goal-based episodes than were those of "poor" students, indicating more persistence on the part of the "good" students. Three data tables and three flowcharts are included. (TJH)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
☑ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
MARGARET RECKER
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

ED318797

TM014923

A Model of Self-Explanation Strategies of Instructional Text and Examples in the Acquisition of Programming Skills

Margaret M. Recker and Peter Pirolli
mimi@Soe.Berkeley.Edu
School of Education
University of California
Berkeley, CA 94720

Paper Presented at
The Annual Meeting of the
American Educational Research Association
Boston, MA

April 18, 1990

1 Introduction

This paper reports studies of students as they learn to program recursive LISP functions in a typical, school-like lesson on recursion containing text and examples and solving a series of programming problems. We are particularly interested in students' learning strategies in new domains. One commonly seen learning strategy when learning from instructional materials is *self-explanation*. In this situation, the student attempts to construct an interpretation of the instructional text and examples prior to solving problems. The types and amounts of self-explanations affect the students' initial understanding and hence have significant impact on their subsequent problem solving performance (Chi, Bassok, Lewis, & Reiman, 1989; Pirolli & Bielaczyc, 1989).

In our learning model (see Fig. 1), the student actively constructs representations of texts and examples based on prior knowledge. This produces a set of example encodings and other relevant domain facts and principles that are stored as declarative knowledge in the learner's memory. During problem solving, upon encountering a partially novel problem, the learner will use as much of his existing domain-specific skill as possible. At problem-solving impasses, in which no previously acquired skills are applicable, the learner resorts to weak-method problem solving. These methods operate on the declarative knowledge acquired from texts and examples. Knowledge compilation mechanisms then summarize each novel problem-solving experience into new domain-specific skills.

We have begun to formalize these self-explanation processes within a computational model in Soar (Laird, Rosenbloom, & Newell, 1986). In our simulation, example explanation is taken to be a process of search in a problem space in which the goal is to generate an explanation structure that satisfactorily interconnects the example to its intended purpose, to already acquired domain knowledge, and to new concepts, facts, principles, etc. that have just been introduced in a lesson.

BEST COPY AVAILABLE

Overview of paper. In the next section, the empirical study, which provides the grounding for the model, is described. In the third section, results of analyses of student elaborations are reported. The fourth section provides a brief overview of Soar, the computational basis for the model and describes how the self-explanation processes have been implemented.

2 The Self-Explanation Study

In our experiment, 12 subjects with little or no programming experience proceeded through a series of lessons on the LISP Tutor (Anderson & Reiser, 1985). The target lesson for our study was the lesson on recursion. In this lesson, the subjects were video-taped and asked to "think aloud" as they read through an instructional booklet prior to solving problems using the Tutor. We also asked subjects to explain the examples to themselves.

The instructional booklet for recursion was carefully crafted for this experiment such that it contains the following components (on separate pages):

1. An abstract description of the structure and function of the components of recursive functions.
2. An example program.
3. A description of the computational behavior generated by recursive functions.
4. A trace of the example program as it processes an input.
5. Some design heuristics for writing recursive functions.
6. A description of how the design heuristics were used in defining the example on page 2.

Pages 1, 3, and 5 are texts, and components 2, 4, and 6 are examples.

After reading through their texts, subjects worked through 12 recursion problems with the LISP Tutor, which were arranged in different sequences for two groups of subjects. The 12 subjects were divided into two groups, *Good* and *Poor*, based on a median split of mean error rates per problem as recorded by the LISP Tutor.

2.1 Protocol Coding Scheme

The verbal protocols were segmented into elaborations, which are pause bounded utterances and are not a first reading of the text. Each elaboration was categorized in terms of the instructional content it refers to and to one of six top level categories. The instructional content was derived from the instructional booklet as follows: each text sentence was assigned a unique proposition number. Each portion of the examples was assigned an example line number.

The top level categories (including protocol examples) are:

- Domain. Statements about programming or recursion (see below).
- Monitor. Statements concerning one's own state of understanding. "I am definitely confused at this point."
- Strategy. Statements about a planned explanation strategy. "I'll look at an example maybe that will help."

- Activity. Statements concerning the task or the instruction. "This paragraph is too long."
- Reread. Statements which are a reread of the text.
- Other. Uncategorized statements.

The domain category, the most important coding category in terms of self-explanation processes, was further decomposed into the following categories (including protocol examples):

- Operation. Statements about the process of the function. "Ok, the first thing it does is null list."
- Result. Statements about the result of a computation. "Cdr gives you the last element of a list"
- Input. Statements about the input to a computation. "When the function gets nil as a value..."
- Structure. Statements about the structure of a function. "You use Cond when you're defining the recursive thing."
- Is-a. Statements relating a particular to a concept. "You're using Carlist as it's own helping function."
- Reference. Statements relating a concept to a particular. "The recursive step is cdr list."
- Purpose. Statements about the purpose of a piece of code. "Cdr... in order to get it closer and closer to the terminating case."
- Analogy. Statements making analogies. "...like n factorial, we did this in math is equal to n times n-1 factorial"
- Entail. Statements describing the entailments of an action. "When the answer is nil then it will stop."
- Plan. Statements about a programming plan. "I see, we keep going and going into easier elements, more and more elementary steps until we know what to code."
- Propose. Statements proposing the meaning of a piece of text or example. "Isn't code the recursives cases the same as assuming the recursive step?"
- Question. Statements questioning the meaning of a piece of text or example . "What is the difference between recursive cases and recursive steps?"

As above, for each of these types of domain elaborations, the content or concept it referred to was also recorded.

Domain elaborations were recorded in these additional two ways. First, any domain elaboration which related the text to an example (or vice-versa) was recorded. We call the phenomenon of relating text and example "making ties." For example, while a student was explaining an example to himself, he explicitly related terms in the example to the previous text page: "(reading) -the tests in the function's condition structure are evaluated- so the zerop and t, zerop and t"

Category	*Group*			
	Text		*Example*	
	Good	Poor	Good	Poor
Domain	84	33	99	41
Monitor	72	18	46	10
Strategy	9	2	8	0
Activity	33	1	18	3
Other	11	2	4	1
Reread	75	56	0	5

Table 1: Summary of elaborations.

Second, the domain elaborations were divided into 2 kinds. We recorded whether the content of the elaboration attended to deeper features of the examples and text, or to the more superficial, syntactic ones.

Finally, sequences of related elaborations were coded into episodes (or macro-codings). We consider a series elaborations related and part of an episode if they are all refer to a similar explanation goal. The episodes were coded with respect to their length and their episode trigger (e.g. a self-imposed goal or a comprehension failure).

3 Analyses and Results

3.1 Comparison to Previous Research

In general, the findings are in agreement with those of Chi et al. (1989) [1]: skill acquisition in a programming lesson is correlated with the quantity and kinds of elaborations made by subjects when they initially try to comprehend the instruction (see Table 1 for a summary of elaborations). More specifically, *Good* students produce a greater total number of elaborations (but not significantly more: $t(10) = 1.46$, $p = 0.09$). However, they do produce significantly more when processing the example ($t(10)=1.93$, $p = 0.04$). Thus, it appears that simply elaborating a great deal does not uniquely determine performance. Rather, some kinds of elaborations are more important than others.

The real difference between groups becomes more evident when looking at the kinds of elaborations made. *Good* students make many more domain elaborations ($t(10)=2.30$, $p = 0.02$). Of these, they make more that attend to the deeper features of the material, and not to the more syntactic ones ($t(10)=1.78$, $p = 0.05$). Finally, the *Good* students make many more elaborations which "make ties." That is, one that attempt to connect the theory explained in the text to the given examples ($t(10) = 2.14$, $p = 0.02$).

In addition, the *Good* students make more strategy statements and seem to be more aware of their own states of comprehension as is reflected by the greater number of these statements ($t(10)=2.53$, $p = 0.02$).

3.2 Additional Results

[1] Preliminary analyses of these data were reported in Pirolli & Bielaczyc (1989).

Category	*Group*			
	Good		*Poor*	
	Total	Mean	Total	Mean
Operation	115	19.16	69	11.50
Result	61	10.16	14	2.33
Reference	22	3.67	7	1.16
Is-a	16	2.67	5	0.83
Structure	8	1.33	2	0.33
Purpose	7	1.16	8	1.33
Analogy	11	1.83	1	0.17
Entail	5	0.83	0	0
Input	6	1	0	0
Plan	4	0.67	7	0.67
Propose	32	5.33	22	3.66
Question	33	5.50	27	4.50
Total	320	53.33	162	27

Table 2: Summary of domain elaborations.

A striking feature of the self-explanation data is the rarity of incorrect elaborations. Although students have much opportunity to draw incorrect generalizations and conclusions, they seldom do. Only 1.8% of all elaborations were judged to be incorrect. This is similar to Siegler & Jenkins' (1989) recent study of how children induce addition strategies. They found that very few incorrect strategies were ever attempted. Thus, it appears making incorrect elaborations is not a key difference between groups. Rather it is simply a failure to elaborate important information.

The episode coding of the self-explanations also yields some interesting results. In general, *Good* students' self-explanations are structured into more episodes (*Good* mean: 5.50; *Poor* mean: 1.17; $t(10)=3.01$, $p = 0.01$). These episodes are either triggered by self-imposed goals (e.g. an attempt to relate the example to the theory explained in the text) or by comprehension failures. This suggests that *Good* students are more persistent and organized in achieving their goals for understanding the instruction.

Finer-grained analyses of domain elaborations are summarized in Table 2. The results show that 40% are related to the *operation* (or process) of LISP functions. Within this category, most elaborations were concerned with how the actual cases of a specific example are evaluated (33%).

The second most common domain category is the *result* category (20%) where the concern is with computing the result of a particular computation. Here, most elaborations involved computing the result of a specific example.

In the reference category (elaborations that relate a concept to an instance, or vice-versa), most involved relating new terms introduced in the text (e.g. terminating-case, recursive-call) to their instantiation in an example. Reference and is-a elaborations accounted for 12% of the elaborations for the *Good* students and 7% for the *Poor*.

The domain elaborations which revealed uncertainty (proposals or questions) were equally divided between the two. They account for 20% of domain elaborations for *Good* students and 30% of domain elaborations for *Poor* students. Not surprisingly, almost all of the uncertain domain

log(1+x)	Factor 1	Factor 2	Factor 3
Activity (total)	.664	-.095	.464
Monitor (total)	.971	-.078	.105
Reread	.03	.071	.947
Strategy (total)	.963	-.061.	.131
Recursion-Related	.925	-.03	-.107
Non Recursion-Related	-.011	.888	-.031
1rst New Errors	-.679	.47	.3
1rst New Time	-.087	.775	.063

Table 3: Factor Analysis: Orthogonal Transformation Solution.

elaboration again concerned the operation (or process) of a LISP function.

A factor analysis of self-explanation data and programming data highlighted some of the subtleties of the relationships between self-explanation and performance. Basically our analysis suggests that the acquisition of new programming skills is associated with the generation of domain elaboration specifically related to the topic of interest (in this case, recursion). The analysis also suggests that gneration of domain elaborations outside the topic of interest may actually be detrimental.

For this factor analysis, we entered subjects' data on several self-explanation and performance measures into a princple components factor analysis. Included in this analysis were counts of subjects' total activity, monitoring, strategy, and rereading elaborations, and subjects' domain elaborations, which were separated into recursion-related and nonrecursion-related elaborations. We also included subjects' errors and latencies on their first opportunity for acquiring a new skill (as indicated by the Lisp Tutor traces). All raw scores were given logarithmic transformations, which yielded factor solutions that were superior to the untransformed scores.

Table 3 presents the weights for the orthogonal factor solution. Three factors were found, with Factor 1 accounting for 56% of the variance, Factor 2 for 25% of variance, and Factor 3 for 19% of the variance. We interpret Factor 1 as capturing the skills associated with the correct identification of comprehension goals and failures, and the generation of useful domain elaborations. From Table 3, it is clear that activity, strategy, and recursion-related elaborations are most heavily associated with Factor 1. Factor 1 also has a heavily negative association with errors on the first opportunity for acquiring a new skill. Factor 2 can be interpreted as capturing the generation of domain elaborations that are less useful (or perhaps, distracting). Factor 2 is heavily associated with the generation of nonrecursion-related domain elaborations, and is positively related to increases in errors and and time in the acquisition of new skills. Factor 3 is somewhat more difficult to interpret. It is strongly associated with amount of rereading, and is positively related to increases in errors on the first opportunity for acquiring a skill. Factor 3 could be interpreted as a measure of unsuccessfull attempts to comprehend the instruction.

4 Overview of the Model

We have begun the implementation of a model of the self-explanation processes in Soar (Laird et al., 1986). Soar is a general problem solving architecture which includes an experience-based learning

mechanism, called *chunking*. Information processing in Soar involves search through problem spaces in order to achieve a particular goal, with knowledge influencing both the structure and efficiency of the search process. Soar has been proposed as a candidate theory of the human cognitive architecture (Newell, in press).

The self-explanation processes are implemented in Soar as a problem-solving process with the goal of creating an explanation structure. Our hypothesis is that an explanation structure consists of a set of interrelated situation models (Kintsch, 1986). Situation models are implemented in Soar as states in the problem space (c.f. Lewis, Newell, & Polk, 1989) . The purpose of these models is to specify (a) the structure and purpose of components of a recursive function, (b) the computational behavior generated by a recursive function, and (c) design heuristics which specify future programmer behavior.

Understanding the instructional material involves two main components: understanding the text and understanding the examples. The instructional text is represented as a set of propositions defining the main principles, facts, and concepts of the target domain. The example provides a concrete instantiation of the theory. However, for a novice programmer, many of the concepts are unfamiliar.

4.1 The Representational Scheme

Each page of the instruction is encoded in working memory as a set of propositions. The propositions represent an ideal model encoding of the declarative text base. Each proposition matches a sentence or sentence fragment in the instruction. This encoding mirrors the encoding of the instruction used in the protocol analysis. The examples are encoded as one working memory element per chunk of code, where a chunk of code depends on the presumed familiarity of the structure[2] to the idealized student.

4.2 Major Problem Spaces

The top level problem space is called *process-instruction*. Within the top space are the two major operators for processing text and for processing examples. These operators are implemented in their own spaces, *read-text* and *explain-example*.

In the *read-text* problem space, textual information is processed with a knowledge of a particular focus. The focus is indicated by the current topic as specified in the title of the page of instruction (e.g. structural information, process information, etc). Depending on the focus, the relevant situation model is constructed and augmented. Impasses occur when an unfamiliar term is encountered. To overcome such impasses, the system can either attempt to ground the term in the examples (if available), flag it as unknown, or ignore it.

Example understanding is implemented in the *explain-example* problem space (see Fig. 1). Here, the system can use knowledge available from prior lessons and from the just-read instructional texts to generate a model of a presented example. For the example understanding process to be effective with respect to future problem solving, it must produce chunks that contain knowledge that will be useful in problem solving. Additionally, the chunks must be retrieved and assembled by cues that will be present in the problem solving context.

[2]So, for example, "Defun function <parameter>" is encoded as one element since a certain degree of familiarity for function definitions can be presumed. In contrast, each sublist in the recursive case in the example program is encoded as one chunk since each involve new concepts.

Similar to Kintsch (1986), we have modeled example encoding as a process that may involve up to three levels. The first level, *verbatim*, is an elaboration of the Soar data chunking technique (Rosenbloom, Laird, & Newell, 1987) for declarative data storage. In data chunking, existing knowledge is retrieved and matched against new information from the outside world and summarized in new productions. These productions can recreate the information at recall time given similar situation cues.

At the second level, *understanding*, the example is parsed with the goal of meeting the recursive function's specifications. This process occurs by bringing to bear previous LISP knowledge and by recognizing matches in the code to the given function's specifications. Finally, at the last level, *learning*, the system attempts to relate elements of the example to concepts introduced in the text.

4.3 The Task

Figure 3 shows how we expect the system to run. As mentioned above, the system's working memory is initialized with the textual propositions representing the first page of the instruction. The system also has knowledge of the instructional topic, i.e. it knows that the text gives a description of the structure of recursive functions. Very little processing occurs at this stage.

The system then has the goal of processing the second page of the instruction, the example recursive function. Here, as explained above, the example is parsed at three levels. For each successful match, the relevant situation model is augmented.

The third page of instruction, a textual description of recursive function evaluations, is parsed with the same mechanisms as the first page. For each text proposition, if no relevant background knowledge exists, the system attempts to match to the example. Again, for each successful match, the functional situation model is augmented. If the proposition cannot be related to the example, it is flagged as unknown.

The situation models created, the way they are related to prior knowledge, and the manner in which they were generated at comprehension time all determine their availability, paths of accessibility, and usefulness at programmer coding time. Pirolli (in press) suggests some aspects of the created situation models that may be relevant to overcoming problem solving impasses in programming. These include the amount of structural information, and information about relevant goals and plans.

5 Conclusion

We have described a Soar computational model of self-explanation strategies of instructional text and examples, based on empirical results. The empirical analyses showed a strong correlation between types and contents of elaborations made and subsequent problem solving performance. In particular, the self-explanations of *Good* students are much more structured into goal-based episodes indicating more persistence on the part of the student. *Good* students make significantly more domain elaborations, especially ones dealing with deeper domain features and ones making ties between the text and examples. Additionally, effective monitoring seems to be tightly integrated into the process.

The computational model describes the space of possible explanation processes and products. In the model, self-explanations are viewed as the process by which a declarative explanation structure is constructed. The explanation structure consists of a set of interrelated situation models which

specify the different aspects of programming. With each additional component of the instruction, the situation models can be updated.

The formalization of self-explanation strategies contributes to our understanding of learning strategies in complex problem-solving domains. Our model addresses the way in which situation models are refined and elaborated by instruction thereby deepening our understanding of learning strategies of typical instructional material. These results should inform research in the area of student modeling in intelligent tutoring systems. In addition, it extends work in artificial intelligence and Soar to more substantial psychological phenomena.

In the future, we plan to further investigate the correlation between self-explanation strategies and problem solving. In particular, we plan to study the role that different instructional representations might have on the effectiveness and efficiency of learning strategies in declarative knowledge acquisition tasks.

References

Anderson, J. and Reiser, B. (1985). The LISP Tutor. *Byte*, 10:159–175.

Chi, M., Bassok, M., Lewis, M., Reiman, P., and Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13:145–182.

Kintsch, W. (1986). Learning from text. *Cognition and Instruction*, 3:87–108.

Laird, J., Rosenbloom, P., and Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning. An Artificial Intelligence Approach*, 1:11–46.

Lewis, R., Newell, A., and Polk, T. (1989). Toward a Soar theory of taking instructions for immediate reasoning tasks. In *Proceedings of the Annual Conference of the Cognitive Science Society.*

Newell, A. (in press). *Unified Theories of Cognition.* Harvard University Press, Cambridge, MA.

Pirolli, P. (in press). Effects of examples and their explanations in a lesson on recursion: A production system analysis. *Cognition and Instruction.*

Pirolli, P. and Bielaczyc, K. (1989). Empirical Analyses of Self-Explanation and Transfer in Learning to Program. In *Proceedings of the Cognitive Science Society Conference.*

Rosenbloom, P. S., Laird, J. E., and Newell, A. (1987). Knowledge-level learning in Soar. In *Proceedings of the National Conference on Artificial Intelligence*, pages 499–504.

Siegler, R. (1989). *How Children Discover New Strategies.* Lawrence Erlbaum, Hillsdale, NJ.

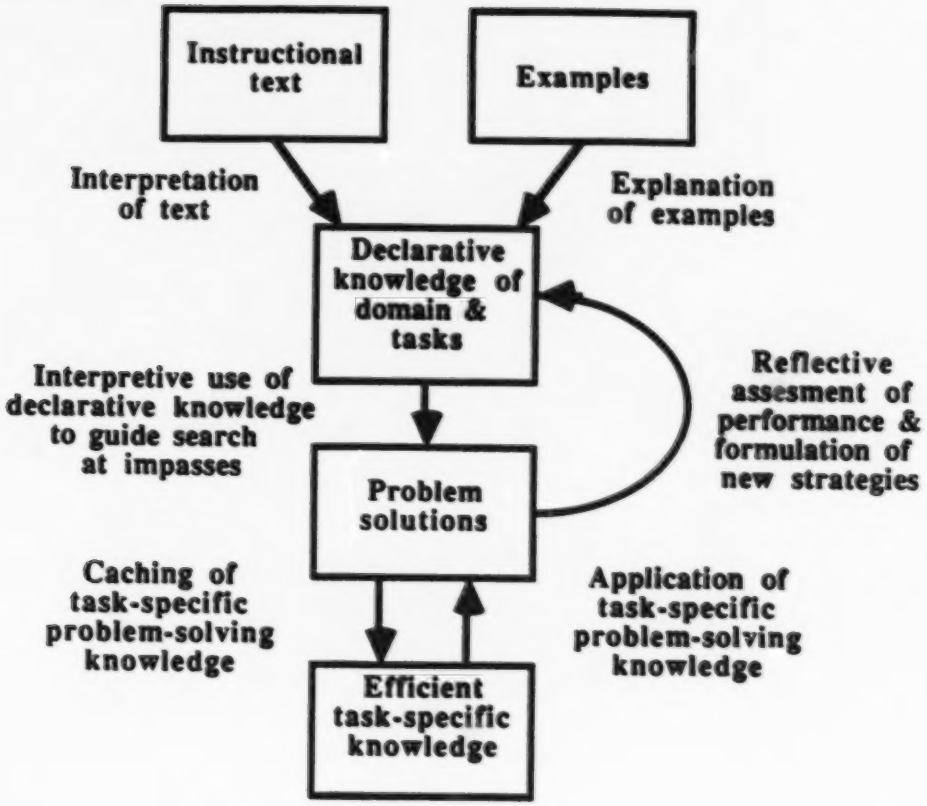
Instructional text
Examples
Interpretation of text
Explanation of examples
Declarative knowledge of domain & tasks
Interpretive use of declarative knowledge to guide search at impasses
Reflective assesment of performance & formulation of new strategies
Problem solutions
Caching of task-specific problem-solving knowledge
Application of task-specific problem-solving knowledge
Efficient task-specific knowledge

Figure 1

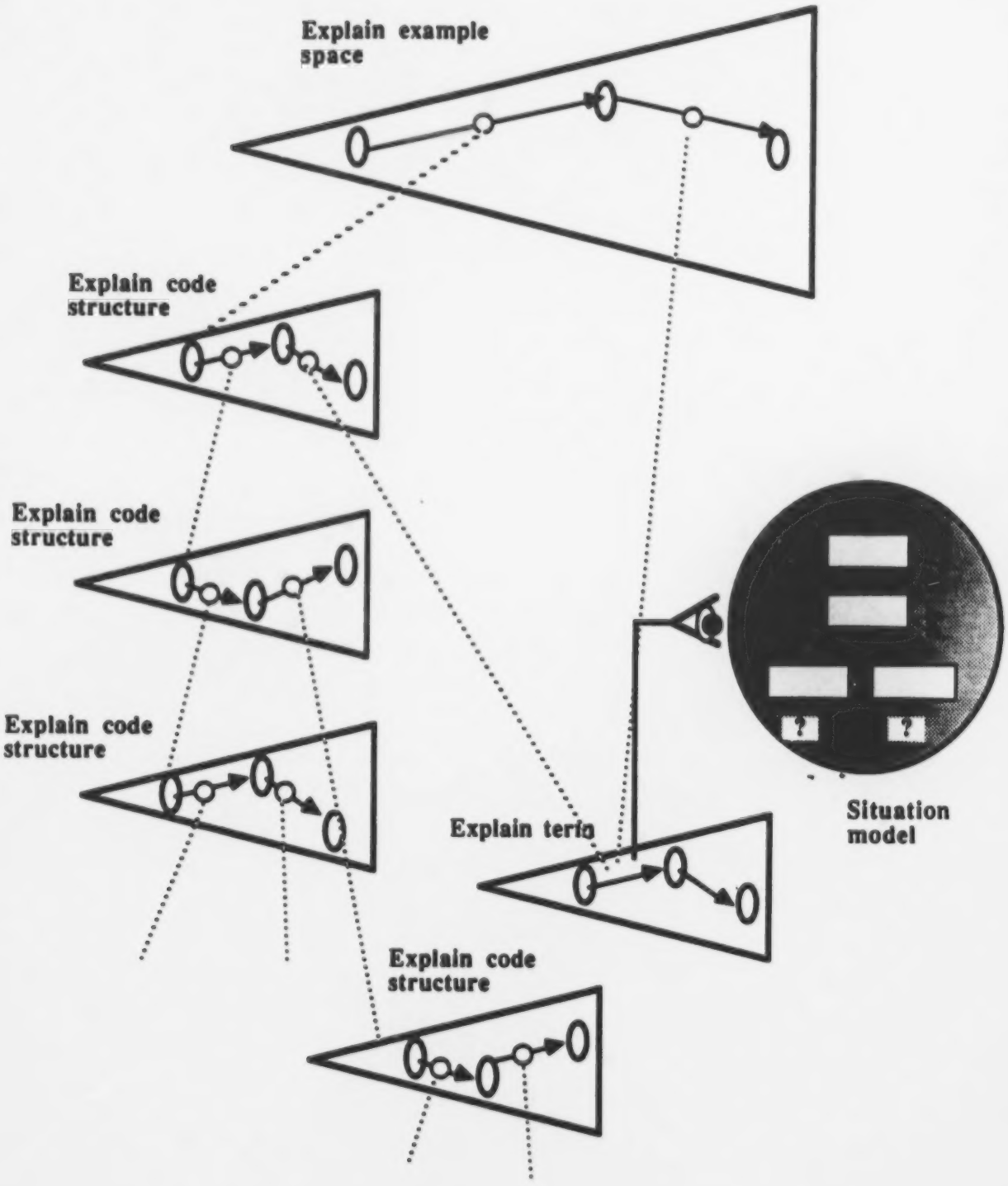
Explain example space
Explain code structure
Explain code structure
Explain code structure
Explain code structure
Explain terms
Situation model

Figure 2

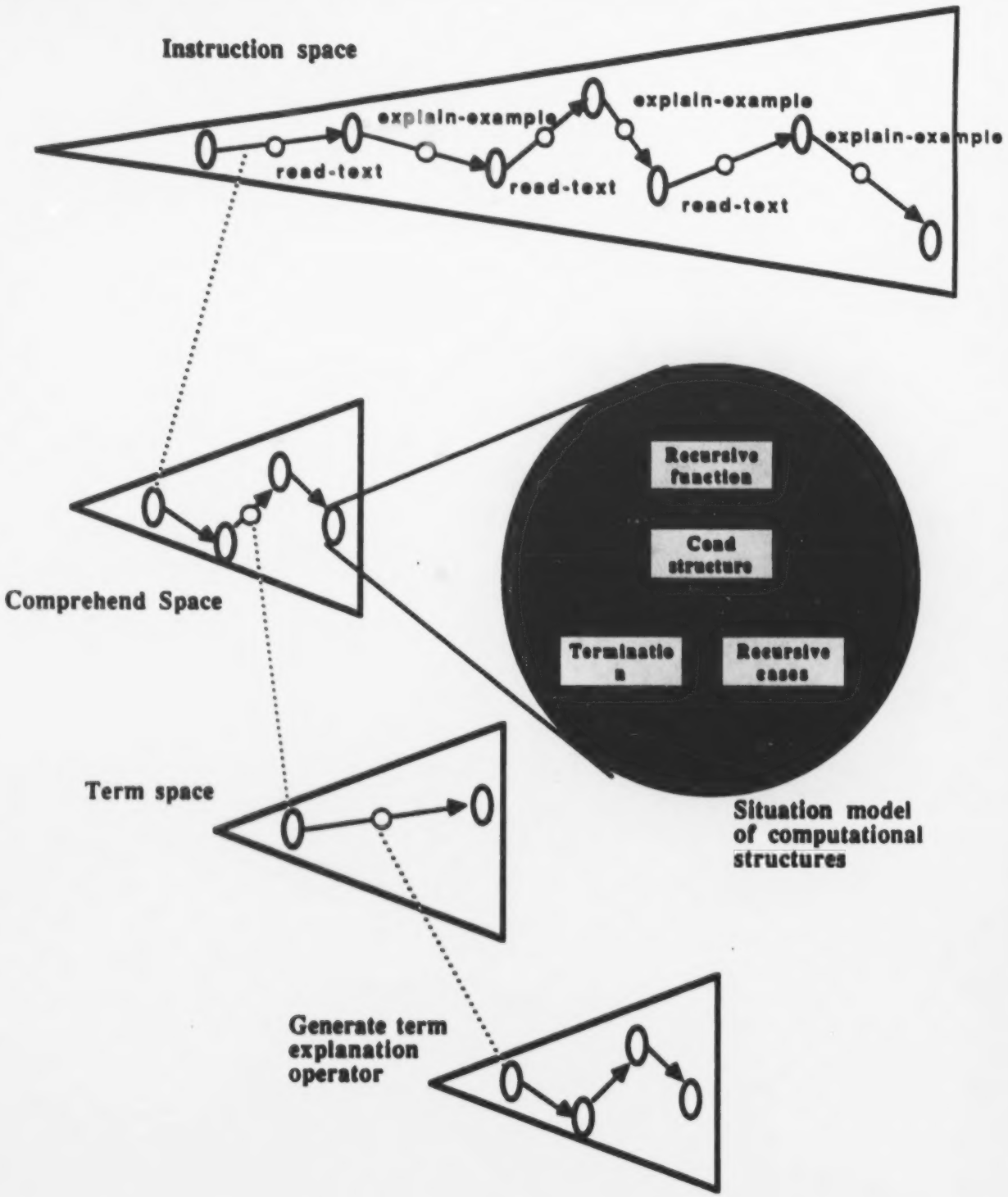
Instruction space
explain-example
explain-example
explain-example
read-text
read-text
read-text
Comprehend Space
Recursive function
Cond structure
Terminatio n
Recursive cases
Situation model of computational structures
Term space
Generate term explanation operator

Figure 3